Tolerating Hardware Faults In Commodity Software: Problems, Solutions, and a Roadmap

Karthik Pattabiraman
(http://blogs.ubc.ca/karthik),
Electrical and Computer Engineering
University of British Columbia (UBC)
Motivation: Hardware Errors

• Errors are becoming more common in processors
  – Soft errors and device variations (timing errors)
  – Processors experience wear-out and thermal hotspots

Source: Shekar Borkar (Intel) - Stanford talk in 2005
Hardware Errors: Traditional Solutions

- **Guard-band**ing
  
  Guard-band**ing wastes power as gap between average and worst-case widens due to variations

- **Duplication**
  
  Hardware duplication (DMR) can result in 2X slowdown and/or energy consumption
An alternative approach

User interacts with the application

Allow errors across the hardware-software boundary, but make sure user does not perceive it.
Why do software techniques work?

Impactful Errors
Software Techniques

Leverage the properties of the application to provide targeted protection, only for the errors that matter to it

Targeted protection mechanisms
Outline

• Motivation

• Techniques developed by my group [DSN’13][CASES’14]

• A brief history of software techniques

• Adoption in Industry

• Research opportunities and roadmap
EDCs: Soft Computing Applications

- Applications in machine learning, multimedia processing
- Expected to dominate future workloads [Dubey’07]

Original image (left) versus faulty image: JPEG decoder
EDCs: Egregious Data Corruptions

- Large or unacceptable deviation in output

EDC image (PSNR 11.37) Vs. Non-EDC image (PSNR 44.79)
EDCs: Goal

- Selectively detect EDC causing faults, but not others

Diagram:
- Detector
- Application Execution
- EDC
- Non-EDC
- Benign

Steps:
1. a → b → EDC
2. c → d
3. e → f
4. g → store
5. checker
6. store
7. g'
EDCs: Fault model

• Transient hardware faults
  – Caused by particle strikes, supply noise

• Our Fault Model
  – Assume one fault per application execution
  – Processor registers and execution units
  – Memory and cache protected with ECC
  – Control logic protected with other methods
EDCs: Main Idea

Our prior work: EDCs are caused by corruption of a small fraction of program data [Flikker - ASPLOS’11]

This work: Critical data can be identified using static and dynamic analysis, without any programmer annotations
EDCs: Initial Study

- Correlation between program data use & fault outcome

Monitor Control/Pointer Data

- Instrument code
- Fault Injection

Performed using LLFI fault injector [DSN’14], at the LLVM IR code level
EDCs: Initial Study

![Bar Chart]

- CRASH: 23%
- EDC: 6%
- Non-EDC: 43%
- BENIGN: 28%

Legend:
- NonPointer NonControl
- Pointer Non Control
- Control Non Pointer
- Control Pointer
void conv422to444(char *src, char *dst, int height, int width, int offset)
{
    for(j=0; j < height; j++) {
        for(i=0; i < width; i++) {
            im1 = (i < 1) ? 0 : i - 1
            ...
            ...
        }
        if (j + 1 < offset) {
            src += w;
            dst += width;
        }
    }
}
EDCs: Algorithm

- Application Source Code
- Compiler
- IR
- EDC Ranking Algorithm
- Selection Algorithm
- Performance Overhead
- Representative inputs
- Data Variables or Locations to Protect
- Backward slice replication

Initial Study ➔ Heuristic ➔ Algorithm
EDCs: Detection Coverage

\[ EDC \text{ Coverage} = \frac{\text{Number of Detected EDCs}}{\text{Total Number of EDCs}} \]

Average EDC Coverage of 82% at 10% performance overhead

Higher is better
EDCs: Selectivity

Average Benign and Non-EDC Coverage of 10 to 15% for overheads from 10 to 25%

Lower is better
SDCTune: Silent Data Corruption (SDCs)

**SDC Output**

**Correct output**

**Results lost:**

Fault occurs → Error activated → Error Masked → Program Finished → Crash/Hang → Benign → SDC
SDCTune: Goals

- **Protecting critical data in soft-computing applications from EDCs**
  - Can we extend this to Silent Data Corruptions (SDCs) in general-purpose applications?

- **Challenge:**
  - Not feasible to identify SDCs based on the amount of data affected by the fault as was the case with EDCs
  - Need for comprehensive model for predicting SDCs based on static and dynamic program features
SDCTune: Main Idea

• Start from Store and Cmp instructions and go backward through program’s data dependencies

• Use *machine learning (CART)* to predict the SDC proneness of Store and Cmp instructions
  – Extract *the related features* by static/dynamic analysis
  – Quantify the effects by classification and regression
  – Estimate SDC rates of different Stores and Cmp instructions
SDCTune: Example Model

Linear Regression for SDC-proneness
SDCTune: Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Benchmark suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>Integer sorting</td>
<td>NAS</td>
</tr>
<tr>
<td>LU</td>
<td>Linear algebra</td>
<td>SPLASH2</td>
</tr>
<tr>
<td>Bzip2</td>
<td>Compression</td>
<td>SPEC</td>
</tr>
<tr>
<td>Swaptions</td>
<td>Price portfolio of swaptions</td>
<td>PARSEC</td>
</tr>
<tr>
<td>Water</td>
<td>Molecular dynamics</td>
<td>SPLASH2</td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate gradient method</td>
<td>NAS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Benchmark suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lbm</td>
<td>Fluid dynamics</td>
<td>Parboil</td>
</tr>
<tr>
<td>Gzip</td>
<td>Compression</td>
<td>SPEC</td>
</tr>
<tr>
<td>Ocean</td>
<td>Large-scale ocean movements</td>
<td>SPLASH</td>
</tr>
<tr>
<td>Bfs</td>
<td>Breadth-First search</td>
<td>Parboil</td>
</tr>
<tr>
<td>Mcf</td>
<td>Combinatorial optimization</td>
<td>SPEC</td>
</tr>
<tr>
<td>Libquantum</td>
<td>Quantum computing</td>
<td>SPEC</td>
</tr>
</tbody>
</table>
SDCTune: Evaluation Method

- Training phase
- Usage Phase
- Evaluation Phase

Features extracted based on heuristic knowledge from training programs
P(SDC|I) from training programs
Training (CART Method) P(SDC|I) Predictor
Estimate the SDC proneness of different program instructions
Find the set of instructions for an overhead bound (∑P(I))
Random Fault Injection Results from testing programs
Actual SDC coverage for testing programs
Features extracted from testing programs

Evaluation Phase
SDCTune: Model Validation

<table>
<thead>
<tr>
<th></th>
<th>Training programs</th>
<th>Testing programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank correlation*</td>
<td>0.9714</td>
<td>0.8286</td>
</tr>
<tr>
<td>P-value**</td>
<td>0.00694</td>
<td>0.0125</td>
</tr>
</tbody>
</table>

Graph showing the rank of overall SDC rates by fault injection experiment for training and testing programs.
SDCTune: SDC Coverage

Training programs:

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>44.8%</td>
</tr>
<tr>
<td>20%</td>
<td>78.6%</td>
</tr>
<tr>
<td>30%</td>
<td>86.8%</td>
</tr>
</tbody>
</table>

Testing programs:

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>39%</td>
</tr>
<tr>
<td>20%</td>
<td>63.7%</td>
</tr>
<tr>
<td>30%</td>
<td>74.9%</td>
</tr>
</tbody>
</table>
SDCTune: Full Duplication and Hot-Path Duplication Overheads

<table>
<thead>
<tr>
<th>Normalized Detection Efficiency</th>
<th>10% overhead</th>
<th>20% overhead</th>
<th>30% overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training programs</td>
<td>2.38</td>
<td>2.09</td>
<td>1.54</td>
</tr>
<tr>
<td>Testing programs</td>
<td>2.87</td>
<td>2.34</td>
<td>1.84</td>
</tr>
</tbody>
</table>
EDCs and SDCTune: Summary

• Software level techniques for tunable and selective protection from EDCs and SDCs
  [DSN’13][DSN’14][CASES’14][TECS1][TECS2]

• Completely automated – no programmer intervention or annotations are needed

• Significant efficiency gain over full duplication
Outline

• Motivation

• Techniques developed by my group [DSN’13][CASES’14]
  • A brief history of software techniques

• Adoption in Industry

• Research opportunities and roadmap
History of S/W techniques: Pre-2000

• Long history of software techniques for high reliability systems going back to IBM MVS, Tandem Guardian
  – Relied on architectural support from the hardware
  – Assumed software was written in transactional style

• Algorithm Based Fault Tolerance – 1984 [Huang and Abraham]: specialized applications in linear algebra

• Many control-flow checking techniques from 1980’s
  – Only protected the program’s control-flow instructions
History of S/W techniques: 2000-2005

• Soft error problem [Sun Server – Baumann 2000]
• ARGOS project from Stanford (McCluksey, 2001)
  – EDDI – software-based instruction duplication
  – CFCSS – Lightweight control-flow checking
• Reliability and Security Engine (RSE) from UIUC (2004)
  – Targeted checking of application properties at runtime
• SWIFT from Princeton (2005)
  – Low-overhead checking through compiler optimizations
• First SELSE workshop launched (2005)
  – Focus on entire system spanning software and hardware
SELSE Papers (2009-2017)

Papers at SELSE 2009-2017 (source: SELSE website)

Data unavailable for years 2005 to 2008. Based on title and abstracts only.
History of S/W techniques: 2010-today

• **Cross-Layer Resilience becomes a buzz word:** Many groups working on this problem including ours

• **Multiple domains:** HPC systems, Embedded Systems

• **Calls from different funding agencies** (DoE, NSF, etc.) for Cross-Layer Resilience Techniques – white papers

• **Conjoined twin:** Approximate Computing takes off
  – Papers at top PL/architecture conferences
Outline

• Motivation

• Techniques developed by my group [DSN’13][CASES’14]

• A brief history of software techniques

• Adoption in Industry

• Research opportunities and roadmap
What about Software Researchers?

• Papers in the top software engineering/testing/reliability conferences about hardware faults and errors over the last 10 years (2006 onwards)
  – **ICSE**: 5 papers (IEEE DL)
  – **FSE**: 6 papers (ACM DL)
  – **ASE**: 7 papers (ACM DL)
  – **ISSTA**: 3 papers (ACM DL)
  – **ICST**: 2 papers (IEEE DL)
  – **ISSRE**: 10 papers (IEEE DL)
  – **Total**: 33 out of over 3000 papers (about 1%)
Example conversations with Software Developers in Industry

- Developer 1 (large s/w company you’ve heard of)
  - Me: How do you handle hardware faults?
  - D1: Do these even occur in the real world?
  - Me: Showing him data gathered by his own company on h/w faults
  - D1: Hmmm… sounds like a problem for QA folks. We don’t deal with faults.

- Tester 1 (large s/w-h/w company you’ve heard of)
  - Me: How do you handle hardware faults?
  - T1: Our hardware folks put in various mechanisms such as ECC memory to mask these
  - H/w guy: Not really, we don’t handle everything
  - T1: Oh, well – that’s not part of our requirements doc. Maybe if we meet our bug targets...
Software Developers

• Most software developers (and testers) ignore hardware faults, or assume faults will be handled by hardware (e.g., ECC memory)

• Even if they recognize the importance of the problem, many think it’s not their problem
  – QA or testing people should take care of it
  – Not part of requirements/specification document
Should we care about developers?

Ultimately, developers are the ones who drive adoption and assimilation within the broader software ecosystem.
Barriers to Adoption: Possible Reasons

• **Reason 1:** Software developers don’t care about anything to do with hardware

• **Reason 2:** Too much time and effort – many other priorities in software development

• **Reason 3:** Lack of high-level abstractions

• **Reason 4:** No easy-to-use tools that integrate with the software development workflow
Barriers to Adoption: Reason 1

• Software engineers don’t care about anything to do with hardware

• Not true. Many counter-examples:
  – Parallelism, both coarse and fine-grained
  – Cache conscious data-structures and algorithms
  – Energy efficiency and energy-aware programming
  – Determinism, memory models, etc.
Barriers to Adoption: Reason 2

• Too much time and effort consuming: many other priorities in software development

• Partially true, but not always
  – Many other time-consuming activities are used e.g., continuous testing, static analysis
  – Techniques for tolerating hardware faults don’t need to be time consuming or effort-intensive
Barriers to Adoption: Reason 3

• Lack of high-level abstractions

• My experience: Mostly True
  – Developers want to reason with quantities they’re familiar with (e.g., runtime, defect rates etc.), not necessarily things like FIT rates, or even coverage
  – Need to be able to reason about cost-benefit tradeoffs of different techniques at the abstract level without going into details
Barriers to Adoption: Reason 4

• No easy-to-use tools that integrate with the software development workflow

• My experience: One of the main reasons
  – Need to understand software developers’ workflow and integrate with it – no deviation
  – Must be able to handle legacy code, weird setups, and multiple languages and libraries
  – S/W maintenance accounts for 60% of costs
What we got right (in my opinion)

• Automated workflow, so little to no effort on the part of the programmer was needed

• Abstraction in terms that ordinary programmers can understand (e.g., performance, coverage)
  – Can do better on this front though

• Use of popular open-source tool (LLVM), which is easy to integrate with workflow (in theory)
What we got wrong (in my opinion)

• **Our abstraction was still too low level** – many programmers didn’t understand coverage

• **Legacy code**: LLVM can’t compile old code, inline assembly, customized build systems

• Did not have an easy path to **QA testing** or **software maintenance** in our long term plan
Outline

• Motivation

• Techniques developed by my group [DSN’13][CASES’14]

• A brief history of software techniques

• Adoption in Industry

• Research opportunities and roadmap
Open Challenges

• Need to build software-based techniques that ordinary programmers can reason about

• Need software techniques that can integrate seamlessly with overall software workflow
  – Should not impede QA and testing process
  – Software maintenance should be considered
  – Legacy code and build systems (if relevant)
The Opportunity: Everett Roger’s Model for Disruptive Innovation

We are still in the innovator/early adopter stage - need to cross chasm and move to “early majority”
Potential Research Roadmap

Understand the issues facing software developers and testers in adopting software techniques for tolerating hardware faults

Build techniques to address these issues in a common research framework for the community to avoid effort duplications

Have developers use the framework in actual practice and identify the issues that come up during the use of the framework
Conclusions

• **Software techniques for tolerating hardware faults in commodity systems have mushroomed**
  – Can be tuned based on the needs of the application
  – Can offer significant efficiency over full duplication

• **Unfortunately, the techniques have seen limited adoption in industry & by software community**
  – We, in the SELSE community, all need to address this
  – Emphasis should be on complete software life-cycle
  – Take our message to the software engineering venues
Acknowledgements

My graduate students (7 current, 10 graduated): Anna Thomas, Qining Lu, Layali Rashid, Majid Dadashi, Bo Fang, Jiesheng Wei, Frolin Ocariza, Kartik Bajaj, Shabnam Mirshokraie, Xin Chen, Farid Tabrizi, Saba Alimadadi, Sheldon Sequira, Nithya Murthy, Abraham Chan, Maryam Raiyat, Justin Li

Collaborators and funding agencies (selected ones below)

http://blogs.ubc.ca/karthik/